VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra Informatiky

Absolvování individuální odborné praxe
Individual Professional Practice in the Company

2018/2019 Daniel Šimek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student:	**Daniel Šimek**
Studijní program:	B2647 Informační a komunikační technologie
Studijní obor:	2612R025 Informatika a výpočetní technika
Téma:	Absolvování individuální odborné praxe Individual Professional Practice in the Company
Jazyk vypracování:	čeština

Zásady pro vypracování:

1. Student vykoná individuální praxi ve firmě: Webvalley s.r.o.
2. Struktura závěrečné zprávy:
   a) Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta.
   b) Seznam úkolů zadaných studentovi v průběhu odborné praxe s vyjádřením jejich časové náročnosti.
   c) Zvolený postup řešení zadaných úkolů.
   d) Teoretické a praktické znalosti a dovednosti získané v průběhu studia uplatněné studentem v průběhu odborné praxe.
   e) Znalosti či dovednosti scházející studentovi v průběhu odborné praxe.
   f) Dosažené výsledky v průběhu odborné praxe a její celkové zhodnocení.

Seznam doporučené odborné literatury:

Podle pokynů konzultanta, který vede odbornou praxi studenta.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **doc. RNDr. Petr Šaloun, Ph.D.**

Konzultant bakalářské práce: Bc. Martin Vavrečka

Datum zadání: 01.09.2017
Datum odevzdání: 30.04.2019

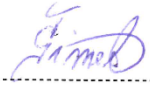
VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA * Fakulta elektrotechniky a informatiky * OSTRAVA 5

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 12. dubna 2019



.....

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.

V Ostravě 12. dubna 2019

webvalley

Webvalley s.r.o.

Adresa:
Pohraniční 504/27
703 00 Ostrava

Kontakt:
info@webvalley.cz
www.webvalley.cz

IČO: 29 44 85 31
DIČ: CZ 29 44 85 31

Tímto bych chtěl poděkovat firmě Webvalley s.r.o, za umožnění vykonávání odborné praxe, a všem zaměstnancům firmy za dobrý kolektiv a vytvoření tak přátelského prostředí. Dále pak děkuji především Ondřeji Boháčovi za ochotné poskytnutí odborné pomoci při řešení praktických úkolů, během absolvování odborné praxe.

Abstrakt

V této bakalářské práci popisuji průběh mé odborné praxe prováděné ve firmě Webvalley s.r.o., ve které jsem působil jako PHP programátor. Představím zde blíže zaměření firmy a pozici, na které jsem byl po dobu vykonávání praxe. Dále zde popíši projekty, kterých jsem se zúčastnil, technologie, se kterými jsem se během praxe setkal, a veškeré zadané úkoly včetně jejich postupu řešení, které zde podrobně popíši. V průběhu odborné praxe jsem pracoval na více projektech, na kterých jsem plnil různé dílčí úkoly. Jako jeden z hlavních projektů mé praxe je interní projekt PageChecker, na kterém jsem se nejvíce podílel na modulu Api Checker a vytvořil zde službu pro kontrolu aplikací. Dalším úkolem byla mimo jiné tvorba administrace v projektu FU OSU Portál studentských prací, ve kterém jsem získal zkušenosti s administrací a mohl jsem je dále uplatnit hned na několika dalších projektech. Dále jsem pak plnil dílčí úkoly na projektech Wave a TONAK. Na závěr mé práce uvedu scházející a získané znalosti a celkové shrnutí praxe.

Klíčová slova

DataTable, Doctrine ORM, front-end, PHP, Symfony, Služby, Webvalley s.r.o

Abstract

In this bachelor thesis I describe the course of my professional experience in the company Webvalley s.r.o., where I‘ve worked as a PHP programmer. I will introduce here the field of the company and the position I have occupied during my professional experience. In addition, I will describe in detail the projects I have participated in, the technologies I have worked with during the professional experience, and all the tasks, including their description of procedure and solutions. During my professional experience I‘ve worked on several projects where I‘ve followed various subtasks. One of the main projects of my professional experience is the internal PageChecker project, where I have participated mostly on Api Checker module and where I have created an application control service. Another task was, among others, the creation of administration in the FU OSU Student Works Portal project, in which I have acquired the experience with administration that I could apply to several other projects. Then I have followed subtasks on Wave and TONAK projects. At the end of my thesis I will present missing and acquired knowledge and a summary of Professional experience.

Key Words

DataTable, Doctrine ORM, front-end, PHP, Symfony, Services, Webvalley s.r.o

Obsah

Obsah ... 7
Seznam použitých symbolů a zkratek ... 9
Seznam obrázků ... 10
Seznam ilustrací a seznam tabulek ... 11
1 Úvod ... 12
2 Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta ... 13
2.1 Historie firmy a její zaměření ... 13
2.2 Pracovní zařazení studenta ... 13
3 Seznam úkolů zadaných studentovi v průběhu odborné praxe ... 14
3.1 Seznámení se systémy firmy a používanými technologiemi ... 14
3.2 Práce na projektu TONAK a.s. ... 14
3.3 Práce na projektu Wave ... 14
3.4 Práce na projektu OSU Portál studentských prací ... 14
3.5 Práce na interním projektu PageChecker ... 15
3.5.1 Modul Pixel Checker ... 15
3.5.2 Modul Api Checker ... 15
4 Zvolený postup řešení zadaných úkolů ... 16
4.1 Seznámení se systémy firmy a používanými technologiemi ... 16
4.1.1 HTML ... 16
4.1.2 CSS ... 16
4.1.3 SQL ... 16
4.1.4 Composer ... 16
4.1.5 GIT ... 16
4.1.6 PHP ... 16
4.1.7 JavaScript ... 17
4.2 Práce na projektu TONAK a.s. ... 17
4.3 Práce na projektu Wave ... 19
4.3.1 Refaktoring a update na PHP 7.1 ... 19
4.3.2 Správa AWS účtů ... 20
4.4 Práce na projektu OSU Portál studentských prací ... 23
4.4.1 Vytvoření náhodného pozdravu ... 23
4.4.2 Omezení na 15 minut přihlášení ... 25
4.4.3 Vytvoření administrace ... 27
4.5 Práce na interním projektu PageChecker ... 29

4.5.1 Modul Pixel Checker 30
4.5.2 Modul Api Checker 30
5 Teoretické a praktické znalosti a dovednosti získané v průběhu studia uplatněné studentem v průběhu odborné praxe 38
6 Znalosti či dovednosti scházející studentovi v průběhu odborné praxe. 39
7 Závěr 40
Literatura 41

Seznam použitých symbolů a zkratek

AJAX	– Asynchronous JavaScript and XML
API	– Application Programming Interface
AWS	– Amazon Web Services
CSS	– Cascading Style Sheets
DOM	– Document Object Model
FU	– Fakulta umění
GIT	– Version Control System
Gitlab	– Git-repository hosting service
HTML	– Hyper Text Markup Language
HTTP	– Hypertext Transfer Protocol
HTTPS	– Hypertext Transfer Protocol Secure
LDAP	– Lightweight Directory Access Protocol
MVC	– Model–View–Controller
ORM	– Object-relational mapping
OSU	– Ostravská univerzita
PDF	– Portable Document Format
PHP	– Hypertext Preprocessor
SQL	– Structured Query Language
URL	– Uniform Resource Locator

Seznam obrázků

Obrázek 1: Ukázka vytvořené administrace a přehledu Amazon účtů..... 19
Obrázek 2: Náhled na vytvořenou administraci s Api Check záznamy 36
Obrázek 3: Ukázka stránky podrobněji daného ApiChecku řešené pomocí tabů 37

Seznam ilustrací a seznam tabulek

Výpis 1: Ukázka Controlleru frameworku Symfony 17
Výpis 2: Ukázka aplikování metody animate() a slideToggle() v jQuery..... 18
Výpis 3: Ukázka twig šablony s cestou k překladům..... 18
Výpis 4: Ukázka career.cs.yml souboru s českou verzí překladu a strukturou klíčů 19
Výpis 5: Ukázka povolení kontroly typů 20
Výpis 6: Ukázka nastavení datových typů 20
Výpis 7: Ukázka vypsání šablony s Amazon účty pomocí cyklu for. 21
Výpis 8: Ukázka načtení Amazon účtů z databáze do entity a následné předání do šablony 21
Výpis 9: Ukázka akce editace účtu. 22
Výpis 10: Ukázka přidání listu do form typu Projektu. 22
Výpis 11: Implementace RandFunction 23
Výpis 12: Zaregistrování RandFunction v config.yml 24
Výpis 13:Ukázka repozitáře s metodou pro náhodný pozdrav..... 24
Výpis 14: Ukázka GreetingExtension 25
Výpis 15: Ukázka řešení SessionListener, který kontroluje neaktivitu..... 26
Výpis 16: Ukázka vytvoření, nastavení a přidání sloupců ve třídě UserDatatable 28
Výpis 17: Ukázka vytvoření UserDatatable ve funkci indexAction()..... 29
Výpis 18: Ukázka pravidla ověření stáří data pomocí parametru a funkce DateInterval() 31
Výpis 19: Ukázka rozšíření třídy ProblemHistory o potřebné vytvoření záznamu..... 32
Výpis 20:Ukazká funkce fetchData() služby BaseApiService..... 33
Výpis 21: Ukázka funkce handleData() 34
Výpis 22: Ukázka funkce makeCheck()..... 35

1 Úvod

Tato bakalářská práce popisuje mé působení ve firmě Webvalley s.r.o, kde jsem vykonával odbornou praxi. Pro absolvování tohoto typu bakalářské práce jsem se rozhodl především z důvodu kladených nároků zaměstnavatelů, kteří upřednostňují především praktické dovednosti, nad teoretickými znalostmi. Bakalářská práce formou odborné praxe mi tedy přišla jako dobrá možnost, jak získat praktické zkušenosti již během studia a zároveň si ověřit a rozšířit již získané teoretické znalosti. Dále pak prohloubit komunikační schopnosti, práci pod tlakem vedoucích nebo časového termínu, a především spolupráci v týmu.

Do firmy Webvalley s.r.o jsem nastoupil na pozici PHP programátor neboli vývojář webových aplikací. V průběhu vykonávání praxe jsem pracoval na několika projektech, na kterých jsem dostával dílčí úkoly. Z počátku se bude jednat o seznámení se s technikami a úpravami vzhledu na projektu TONAK za použití HTML, CSS a JavaScriptu. Následně seznamování se s programovacím jazykem PHP a frameworkem Symfony na projektu Wave, na kterém následně provedu update na novější verzi PHP včetně refaktorování. Postupem času se dostanu k řešení obsáhlejších úkolů na projektu FU OSU Portál studentských prací. Bude se jednat o portál Umělecké fakulty OSU, na který budou studenti nahrávat své práce. Kromě menších úkolů, jako je příprava služby pro ověření neaktivity, nebo služby náhodného pozdravu, bude má práce směřovat převážně na tvorbu administrace pro správu uživatelů a stránek. Tyto zkušenosti s administrací dále využiji ve více projektech.

Hlavním cílem praxe bude práce na interním projektu PageChecker. Projekt je rozdělen do dvou modulů, kde většina úkolů bude plněna na jeho části Api Checker. Úkolem bude mimo jiné vytvořit službu pro kontrolu projektů a aplikací, na základě zaslaných informací z daného aplikačního rozhraní kontrolovaného projektu. Následně bude třeba získaná data kontrolovat a případně ověřit správnou funkčnost dle stanovených pravidel.

V následujících částech bakalářské práce představím společnost a přiblížím pracovní pozici vykonávanou po dobu praxe. Následně v bodech uvedu používané technologie, se kterými jsem se během práce setkal. Hlavní pozornost je kladena představení projektů, zadaným úkolům, a především postupu řešení, které jsem aplikoval pro splnění zadaných úkolů. Na závěr této práce uvedu zhodnocení získaných praktických dovedností, teoretických znalostí a celkový přínos praxe.

2 Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta

2.1 Historie firmy a její zaměření

Společnost Webvalley s.r.o [1], sídlící ve Vítkovicích v Ostravě, vznikla 26. 6. 2012. Firma se zabývá především tvorbou webových stránek na míru dle potřeb zákazníka a vyvíjením mobilních aplikací a jejich optimalizací. Dále pak nabízí uživatelské testování a A/B testování pro zvýšení konverze. Velkou míru zakázek tvoří práce se zákazníky v zahraničí, konkrétně v USA.

2.2 Pracovní zařazení studenta

Po dobu vykonávání mé odborné praxe jsem pracoval na pozici PHP programátor a front-end vývojář. Na začátku praxe jsem pracoval na projektu TONAK, na kterém jsem plnil jednotlivé menší úkoly zaměřené spíše na front-end vývoj, tedy práce s HTML, JavaScript a CSS. Dále jsem pak pracoval převážně s PHP s frameworky jako jsou Symfony nebo Nette. Postupem času jsem se mimo jiné dostal k rozsáhlejším úkolům v podobě vytvoření administrace pro uživatelské účty a stránky na projektu OSU Portál studentských prací. Hlavním úkolem pak bylo vytvoření služby v projektu PageChecker pro kontrolu správného chování nasazených webů a aplikací.

3 Seznam úkolů zadaných studentovi v průběhu odborné praxe

3.1 Seznámení se systémy firmy a používanými technologiemi

Po nástupu do firmy jsem byl zaregistrován do všech potřebných systémů. Jeden z prvních úkolů byl seznámit se s technologiemi, jakou je například webový Git repozitář, a nastudovat si jejich možnosti. Dále jsem se začal seznamovat se skriptovacím jazykem PHP, protože jsem se s tímto jazykem doposud nesetkal. Ve Webvalley je většina projektů založená na frameworcích Symfony nebo Nette. Mým dalším úkolem tedy bylo seznámit se se strukturou a pochopit smysl těchto PHP frameworků pro snadnější řešení následujících úkolů.

3.2 Práce na projektu TONAK a.s.

Během studování struktury a používání frameworku Symfony a Nette, jsem byl zapojen do vyvíjení e-shopu TONAK a.s. Jedná se o e-shop pro firmu zabývající se prodejem a výrobou klobouků. Protože jsem se teprve seznamoval, mé první úkoly byly spíše rutinního typu. Zpočátku se jednalo o samotné zprovoznění projektu, který je nahrán na GitLabu a seznámení se s jeho strukturou. Dále mi pak byly zadávány úkoly, zaměřující se zejména na úpravu front-endu za využití HTML, CSS a JavaScriptu.

3.3 Práce na projektu Wave

Během práce na projektu TONAK jsem se začal také seznamovat s interním projektem Wave. Jedná se o interní firemní projekt určený pro správu jejich projektů. Bude se zaměřovat na jednotlivé části, jako je například poskytování veškerých informací o projektech, tedy informace o produktu, vývojovém týmu, heslech k projektům a další. Dále by pak měl sloužit k nalezení chyb při průchodu nakupovacím procesem (např. pokud se objednávka nespojí s objednávkovým systémem, nahlásí to firmě chybu). Jako další bude poskytovat veškerá analytická data ohledně projektu pro lehké vytváření reportu pro klienty. Z počátku budou úkoly směřovat na seznámení se s frameworkem Symfony v podobě refaktorování a provedení aktualizace PHP na novější verzi. Následně se dostanu k vytvoření administrace pro správu Amazone účtů.

3.4 Práce na projektu OSU Portál studentských prací

Další úkoly byly směřovány na OSU portál. Jedná se o portál pro Ostravskou univerzitu, konkrétně Uměleckou fakultu. Studenti zde budou moct nahrávat své práce ve formátech jako jsou například PDF, dále pak videa, fotografie a podobně. Bude zde napojení na LDAP, což je definovaný protokol pro ukládání a přístup k datům na adresářovém serveru. Na tomto projektu jsem měl za úkol spolupracovat především na administraci uživatelů a stránek. Mimo jiné jsem pak dostával dílčí úkoly jako například ošetřit neaktivitu případným odhlášením nebo vytvořit službu pro náhodný pozdrav.

3.5 Práce na interním projektu PageChecker

Jedná se o interní projekt, který počtem splněných úkolů tvoří hlavní náplň mé bakalářské práce. PageChecker je začínající interní aplikace, která se bude starat o kontrolu business logiky. Součástí této aplikace budou dva moduly.

3.5.1 Modul Pixel Checker

Jedná se o modul, který bude kontrolovat přítomnost daného kusu kódu na stránce. V tomto případě se jedná o větší počet kusů pixelu na prodejních stránkách, které sledují nákup. Smyslem tohoto modulu je předejít nechtěnému smazání pixelů na stránkách tím, že bude neustále kontrolována jeho přítomnost a zda se zobrazují správně. Dále bude sledovat změny stránek, ukládat historii a porovnávat je mezi sebou. V případě chyb nebo větších změn bude zaslána notifikace developerovi pro překontrolování. Úkoly související s tímto projektem spočívaly v přípravě administrace a nachystání seznamu stránek s pixely.

3.5.2 Modul Api Checker

Api Checker je druhý modul projektu PageChecker, který se bude starat o kontrolu aplikací pomocí jejich API. Modul se bude k aplikaci chovat jako k blackboxu, kde implementace Api Checkeru je zcela nezávislá na implementaci dané aplikace. Danou aplikaci bude kontrolovat na změnách parametrů, které dostaneme z API projektu, dle nastavených pravidel. Aplikace bude očekávat, že odpověď API bude ve formátu JSON řetězce. Hlavním úkolem bude připravit službu, která se bude starat o získání JSON řetězce z API a následně kontrolovat získané hodnoty pomocí nastavených pravidel. Dále pak bude potřeba definovat pravidla spolu s ostatními náležitostmi. V poslední řadě připravím administraci pro přehled a správu Api Checker záznamů. Nakonec provedu front-end úpravy týkající se především responzivní administrace a vytvoření přihlašovací stránky.

4 Zvolený postup řešení zadaných úkolů

4.1 Seznámení se systémy firmy a používanými technologiemi

Tato část bude věnována technologiím, se kterými jsem se setkal v průběhu odborné praxe.

4.1.1 HTML

Hypertext Markup Language (HTML) [2] je značkovací jazyk používaný při tvorbě webových stránek.

4.1.2 CSS

Kaskádové styly [3] jsou jazyk, který definuje, jak mají být nastylovány jednotlivé elementy dokumentu HTML. Určuje, jak budou elementy vypadat, a jak se mají zobrazovat v jednotlivých prohlížečích.

4.1.3 SQL

Strukturovaný dotazovací jazyk používaný v relačních databází pro ukládání a manipulaci s daty.

4.1.4 Composer

Tento nástroj [4], který jsem používal na všech projektech, na kterých jsem se podílel, mi byl velkým pomocníkem. Jedná se o nástroj pro správu závislostí v PHP. Composer nám tedy umožňuje pomocí jednoho příkazu deklarovat všechny knihovny, na kterých daný projekt závisí. Stará se jak o stahování a instalaci nových balíčků a knihoven, tak i o následné aktualizace.

4.1.5 GIT

První technologie a zároveň pro mne novou technologií, se kterou jsem pracoval, byl systém zprávy verzí Git [5]. Firma má vlastní server Git postavený na platformě GitLab. Tento systém je důležitou součástí při práci v týmu, jelikož umožňuje přístup k projektům a následné verzování kódu.

4.1.6 PHP

Další technologií, ve které jsem byl nováčkem, je skriptovací jazyk PHP [6]. Tento jazyk je převážně používán při vývoji webových aplikací a dynamických internetových stránek. V tomto případě se veškeré skripty provádějí na straně serveru a k uživatelům přichází už hotový výsledek těchto skriptů. Ve firmě jsem se setkal se dvěma frameworky, kteří rozšiřují PHP v mnoha stránkách.

- **Nette** [7] – Jedná se o open source framework, vyvíjený v České republice. Za vznikem Nette stojí David Grudl. Nyní se o další rozvoj stará Nette Foundation. Framework klade důraz především na bezpečnost, mimo jiné si pak zakládá na čistotě a jednoduchosti kódu a znovu použitelnosti. Prakticky je Nette sada komponent, která dohromady tvoří framework.

- **Symfony** [8] – Většina projektů, na kterých jsem se podílel, byla psána právě pomocí tohoto frameworku. Jedná se o velice rozšířený open source framework, tvořený komponenty, které jsou sadami oddělených a opakovaně použitelných knihoven. Symfony tvoří základní stavební kámen webových aplikací a vychází z návrhového vzoru MVC. Aplikace tedy stojí na třech částech: Kontrolery (řízení), Modely (logika) a Pohledy (výstupy v podobě Twig šablon).

```php
/**
 * Class PixelController
 *
 * @package App\Controller
 * @Route("/pixels")
 */
class PixelController extends Controller
{
    /**
     * @Route("/", name="app_pixel_index")
     * @param Request $request
     * @return \Symfony\Component\HttpFoundation\Response
     */
    public function indexAction(Request $request)
    {
        $isAjax = $request->isXmlHttpRequest();
        $datatable = $this->get('sg_datatables.factory')->create(PixelDatatable::class);
        $datatable->buildDatatable();

        if ($isAjax) {
            $responseService = $this->get('sg_datatables.response');
            $responseService->setDatatable($datatable);
            $responseService->getDatatableQueryBuilder();

            return $responseService->getResponse();
        }

        return $this->render('pixel/index.html.twig', array(
            'pageTitle' => 'Pixels',
            'datatable' => $datatable,
        ));
    }

    ...
}
```

Výpis 1: Ukázka Controlleru frameworku Symfony

4.1.7 JavaScript

JavaScript [9] je skriptovací programovací jazyk, který nám umožňuje vytvořit například dynamicky aktualizovaný obsah na webu, animace, ovládání multimédií, nebo kontrolu formulářů v reálném čase. Jazyk patří do skupiny objektově orientovaných.

- **jQuery** – při práci s JavaScriptem jsem nejčastěji používal jQuery [10]. Jedná se o rozšiřující knihovnu jazyka JavaScript, která se především zaměřuje na zjednodušení manipulace DOM, volání AJAX a odchytávání událostí.

4.2 Práce na projektu TONAK a.s.

Jak jsem již zmínil, jedná se o e-shop pro klienta, který se zabývá prodejem a výrobou klobouků. Celý projekt je psán pomocí frameworku Symfony. Jednotlivé verze jsou uloženy na serveru Webvallley, který je postaven na platformě GitLab.

Následující postup zprovoznění projektu jsem aplikoval u každého dalšího projektu, na kterém jsem pracoval. Začal jsem stažením projektu z GitLabu a nainstalováním jeho veškerých potřebných

balíčků včetně Symfony. Tento krok mi velice ulehčil Composer, který se o vše postaral díky jednomu příkazu. Instalace veškerých závislostí projektu se provede pomocí příkazu *composer install*.

Dalším krokem bylo seznámení se s projektem a jeho strukturou. Jelikož se jednalo o projekt ve finální části, byl velice rozsáhlý. Protože jsem v PHP nováčkem, mé úkoly spočívaly převážně v grafické úpravě stránek, dle klientových požadavků. Ve většině případů se jednalo o přidaní tlačítek a animací na odkazovací tlačítka, animace otevíraní záložek a podobně. Přidání plynulých přechodů jsem řešil pomocí metody, která je součástí jQuery: *animate()*.Tato metoda umožňuje vytvoření a nastavení animace daného elementu. Syntaxe je v podobě *$(selector).animate({params}, speed, callback);*, kde *params* definuje vlastnosti CSS, které mají být animovány. Další metodou, kterou jsem použil opakovaně, je *slideToggle()*, která zobrazuje a skrývá elementy posuvnou animací viz Výpis 2.

```
1.  {% block body_js %}
2.      <script type="text/javascript">
3.          $(".button--js-scroll").click(function() {
4.              $('html, body').animate({
5.                  scrollTop: $("#" + $(this).data("show")).offset().top
6.              }, 1000);
7.          });
8.          $(".positions-label").click(function(){
9.              $(this).parent().children('.positions-content').slideToggle("slow");
10.         });
11.     </script>
12. {% endblock %}
```

Výpis 2: Ukázka aplikování metody *animate()* a *slideToggle()* v jQuery

Zbytek práce spočíval v plnění stránek a přípravě k překladům. Jelikož je web dostupný v anglické a české verzi, bylo zapotřebí zajistit překlad. O to se stará komponenta *Translation* [11], která je součástí Symfony. Vše již bylo připravené, proto má práce spočívala pouze v přípravě Twig šablon a souborů s danými texty. V šablonách bylo kromě struktury zapotřebí definovat obsah pomocí cesty k jednotlivým klíčům s texty pro danou šablonu a specializovaného tagu *trans* viz Výpis 3.

```
1.  ...
2.  <div class="item">
3.      <h2 class="h h_3 item-headline">{{ 'description.first.title'|trans }}</h2>
4.      <ul class="list">
5.          <li class="list-li">{{ 'description.first.list.first'|trans }}</li>
6.          <li class="list-li">{{ 'description.first.list.second'|trans }}</li>
7.          <li class="list-li">{{ 'description.first.list.third'|trans }}</li>
8.      </ul>
9.  </div>
10. ...
```

Výpis 3: Ukázka twig šablony s cestou k překladům

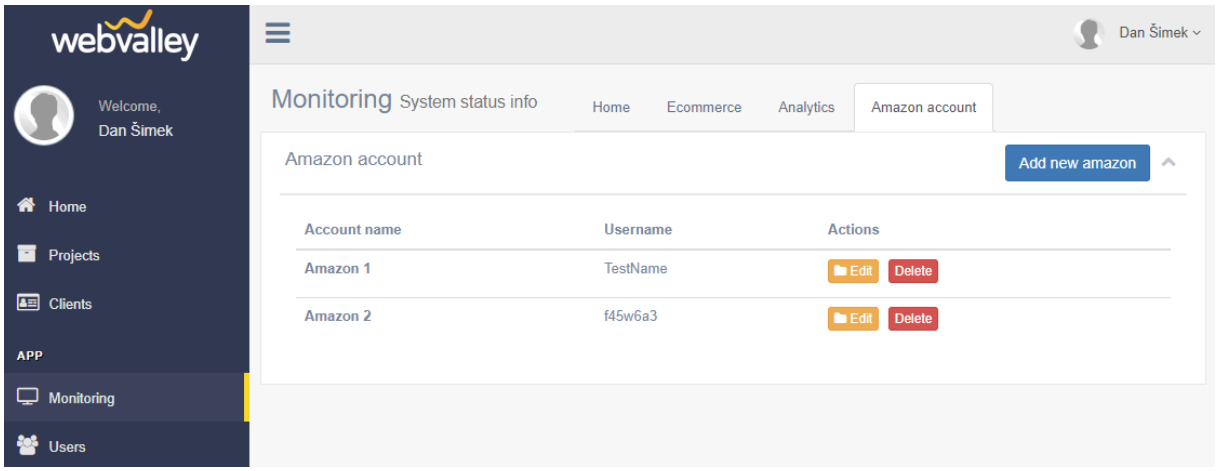
Dané soubory s klíči a texty se musí nacházet v *Resources/translations/*. Velice důležité je pak také jejich správné pojmenovaní, které musí být následujícího typu: *domain.locale.loader*, kde *domain*- je název šablony, *locale*- je lokalita, ve které jsou překlady k dispozici (např en, cs), a *loader*- určuje jaký soubor má Symfony analyzovat a načíst. Příklad pojmenování pro anglickou verzi: *career.en.yml* (překlady pro stránku s kariérou). Tyto jednotlivé soubory jsou strukturované podle klíčů viz Výpis 4. Stejný soubor je vždy nutno vytvořit se stejnou strukturou klíčů, jak pro český, tak i pro anglický jazyk.

```yaml
description:
    first:
        title: 'Koho hledáme?'
        list:
          first: 'Nové kolegy, kteří k nám zapadnou a hlavně ty, kteří chtějí růst s námi'
          second: 'Ty, kteří učí sebe a ostatní'
          third: 'Ty, kteří se chtějí podílet na tvorbě našich klobouků'
    second:
        title: 'Co vám nabízíme?'
        list:
          first: 'Zaškolíme vás na potřebných pracovištích'
          second: 'Práci v neformální a přátelské atmosféře'
          third: 'Kariérní růst podle vašich schopností'
          fourth: 'Pracujeme v jednosměnném provozu, ideální pro maminky s dětmi'
    third:
        title: 'Benefity zaměstnance'
        list:
          first: 'Příspěvek na závodní stravování'
          second: 'Týden dovolené navíc'
          third: 'Péče o vaše zdraví'
          restaurant_notice: '(závodní restaurace ve firmě)'
```

Výpis 4: Ukázka *career.cs.yml* souboru s českou verzí překladu a strukturou klíčů

4.3 Práce na projektu Wave

Jedná se o interní projekt, na kterém jsem měl hned několik úkolů. Zpočátku jsem si opět procházel projekt, který je postavený na frameworku Symfony, abych porozuměl tomu, co zde již bylo naimplementováno a zrychlil tak mou budoucí práci na projektu.



Obrázek 1: Ukázka vytvořené administrace a přehledu Amazon účtů

4.3.1 Refaktoring a update na PHP 7.1

Protože byla v projektu zastaralá verze PHP 5, mým prvním úkolem bylo převést tento projekt na novější verzi PHP 7.1. Při převádění jsem zároveň prováděl refaktoring celého projektu. Tento úkol nebyl nijak složitý, ačkoliv byl velmi náročný na čas, z důvodu rozsáhlosti projektu, neboť obsahoval desítky souborů. Úkol spočíval v procházení veškerých PHP souborů a úpravě kódu, aby byl čitelnější a dle stanovených standardů.

Mezi jednu z největších novinek PHP 7 [12] patří typová kontrola pro skalární datové typy. Ta nám umožňuje kontrolovat předávané datové typy PHP a v případě neplatnosti zobrazí výjimku, jako je např: *Uncaught TypeException: Argument 2 passed to nazevFunkce() must be of the type integer, string given, called in…* Aby byla tato kontrola aktivní, je nutné ji povolit na začátku každého souboru pomocí *declare(strict_types=1)* viz Výpis 5.

```
1. <?php
2. declare(strict_types=1);
3.
4. namespace AppBundle\Controller;
5.
6. use ...
```

Výpis 5: Ukázka povolení kontroly typů

Po aktivování kontroly datových typů v PHP souborech, bylo potřeba upravit veškeré funkce a stanovit datové typy vstupních hodnot. Stejným způsobem lze kontrolovat i návratové typy, díky čemuž se kód stává čitelnějším viz Výpis 6.

```
1.  /**
2.   * @return \DateTime
3.   */
4.  public function getCreatedAt() : \DateTime
5.  {
6.      return $this->createdAt;
7.  }
8.
9.  /**
10.  * @param \DateTime $createdAt
11.  */
12. public function setCreatedAt(\DateTime $createdAt) : void
13. {
14.     $this->createdAt = $createdAt;
15. }
```

Výpis 6: Ukázka nastavení datových typů

4.3.2 Správa AWS účtů

Druhou částí mé práce bylo vytvořit administraci Amazon účtů. Zadáním bylo přidat záložku do již existující šablony a kontroleru Monitoring. V podokně této záložky bude možnost vytvářet nové účty a editovat a mazat již přidané, které budou vypsány v tabulce. Dále pak možnost přiřazení Amazon účtu k projektu.

Na začátku bylo zapotřebí vytvořit novou entitu *AmazonAccount* s atributy: *id*, *createdAt* (datum vytvoření), *name* (název účtu), *username*, *password*, *project*, a *hidden* (pro následné mazání). Entita [13] je ve zjednodušení kontejner na data, který je potřebný jak při komunikaci s databází, tak s vytvářením dalšího záznamu nebo editaci. Každý atribut entity je mapovaný na daný sloupec tabulky uložený v databázi. Proto je zapotřebí vytvořit gettery a settery k daným atributům pro získání a nastavení hodnot.

Nyní bylo potřeba načíst již vytvořené Amazon účty z databáze a předat je do twig šablony, kde jsem připravil tabulku pro vypsání účtů a přiřadil ke každému záznamu tlačítka s akcí smazání a editace. Připravení šablony bylo jednoduché díky podpory cyklů v twig šablonách viz Výpis 7.

```
1.  {% for amazonAccount in amazonAccounts %}
2.      <tr>
3.          <th scope="row">
4.              {{ amazonAccount.name }}
5.          </th>
6.          <td>
7.              {{ amazonAccount.username }}
8.          </td>
9.          <td>
10.             <a href="{{ path('amazon_account_edit_byId', { 'id': amazonAccount.id })
     }}" class="btn btn-warning btn-xs"><i class="fa fa-folder"></i> Edit </a>
11.             <a href="{{ path('amazon_account_delete_byId', { 'id': amazonAccount.id
    }) }}#amazon" class="btn btn-danger btn-xs">Delete </a>
12.         </td>
13.     </tr>
14. {% endfor %}
```

Výpis 7: Ukázka vypsání šablony s *Amazon* účty pomocí cyklu *for*.

K datům v databázi přistupuji pomocí knihovny třetí strany Doctrine ORM [13]. Tento framework umožnuje pracovat s daty jako s objekty. Pomocí Entity Manageru, který je součástí Doctrine ORM, lze jednoduše načítat, mazat či aktualizovat záznamy viz Výpis 8.

```
1.  public function indexAction() : Response
2.  {
3.      $em = $this->getDoctrine()->getManager();
4.      $amazonAccounts = $em->getRepository(AmazonAccount::class)-
    >findBy(array('hidden' => false));
5.
6.      ...
7.
8.      return $this-
    >render(':monitoring:index.html.twig', array('lastCrons' => $lastCrons,
9.              'amazonAccounts' => $amazonAccounts));
10. }
```

Výpis 8: Ukázka načtení *Amazon* účtů z databáze do entity a následné předání do šablony

Mazání dat z databáze je možné dvěma způsoby. První způsob je data trvale smazat pomocí příkazu *DELETE*. Druhý způsob je přidání atributu typu *boolean*, v našem případě *hidden*, kde pouze měníme hodnotu *true* nebo *false* respektive zobrazit nebo skrýt. Tato druhá metoda se používá v případě, že budou data ještě někdy potřeba, jako v našem případě. Proto pomocí metody *findBy(array('hidden' => false))*, která je součástí Entity Manageru, vyselektujeme pouze ty účty, které nejsou smazány respektive skryty viz Výpis 8.

Dále bylo zapotřebí připravit twig šablonu spolu s form typem pro Amazon účet, který jsem následně použil u již zmíněných akcí editace a vytvoření.

Postup při implementování akcí pro přidání nového účtu, upravení již existujícího, a mazání byl podobný. Formuláře jsem vytvořil pomocí integrované komponenty Form [14] v Symfony, která usnadňuje práci s formuláři. Sestavení formuláře jsem provedl pomocí metody *createForm()* a za použití

připraveného form typu viz Výpis 9. Dále pak, po schválení formuláře a validaci formuláře, provedu opět pomocí Doctrine ORM požadovanou akci. V případě vytvoření nového záznamu se používá metoda *persist()*, která je součástí Entity Manageru. Výhoda persistování je ta, že se aplikuje u entity pouze jednou, a to po vytvoření její nové instance. Nadále je už tato instance pod kontrolou Entity Manageru, který se stará o veškeré další změny entity bez nutnosti ukládání při každé změně. Na konci všech provedených změn je nutné je potvrdit a odeslat pomocí metody *flush()*. U editace jsem namísto metody *persist()* použil metodu *merge()*, protože se jedná o editaci již vytvořeného záznamu viz Výpis 9. V případě mazání záznamu jsem pouze nastavil hodnotu atributu *hidden* na *true*.

```
1.  public function editAction(AmazonAccount $amazonAccount, Request $request)
2.  {
3.      $form = $this->createForm(AddAmazonAccount::class, $amazonAccount);
4.      $form->handleRequest($request);
5.      if ($form->isSubmitted() && $form->isValid()) {
6.          $em = $this->getDoctrine()->getManager();
7.          $em->merge($amazonAccount);
8.          $em->flush();
9.          return $this->redirectToRoute('app_monitoring_index');
10.     }
11.
12.     return $this->render('monitoring/form-amazon.html.twig', [
13.         'form' => $form->createView(),
14.         'account' => $amazonAccount,
15.     ]);
16. }
```

Výpis 9: Ukázka akce editace účtu.

Poslední část úkolu bylo vytvořit možnost přiřazení jednotlivých účtů k daným projektům. Protože administrace projektů zde již byla vytvořena, přidal jsem pouze možnost vybrat si z nabídky požadovaný Amazon účet. Bylo zapotřebí upravit entitu projektu a editační formulář, kde jsem přidal otevírací list s dostupnými Amazon účty, které jsem vyselektoval pomocí parametru *query_builder* viz Výpis 10.

```
1.  ->add('amazonAccount', EntityType::class, [
2.      'multiple' => false,
3.      'class' => AmazonAccount::class,
4.      'query_builder' => function (EntityRepository $er) {
5.          return $er->createQueryBuilder('u')
6.          ->where('u.hidden = 0');
7.      },
8.      'choice_label' => 'getName',
9.      'choice_value' => 'id',
10.     'required' => false,
11. ])
```

Výpis 10: Ukázka přidání listu do *form typu* Projektu.

Posledním úkolem bylo přidat možnost mazání uživatelských účtů a možnost přiřazení účtů k projektům. Protože se jednalo o obdobný úkol, jako v případě Amazon účtů, nebylo pro mne již složité jeho splnění.

4.4 Práce na projektu OSU Portál studentských prací

Jeden z hlavních projektů mé bakalářské praxe byl Portál studentských prací pro uměleckou fakultu Ostravské univerzity. Oproti předešlým projektům, které byly již v pokročilém stádiu, se jednalo o spolupráci na teprve začínajícím projektu.

V průběhu vyvíjení projektu jsem dostal více úkolů. Jedním z hlavních úkolů bylo vytvořit administraci pro správu uživatelských účtů. Dále jsem pak dostal dílčí úkoly v podobě vytvoření služby pro pozdravy, nebo ošetření doby přihlášení.

4.4.1 Vytvoření náhodného pozdravu

Prvním a zároveň seznamovacím úkolem bylo vytvořit službu pro pozdravy. Jednalo se o náhodný pozdrav, dle klientových požadavků, kterým se bude zdravit právě přihlášený uživatel. Tyto jednotlivé pozdravy jsou uloženy v databázi. V tomto úkolu jsem se naučil implementovat službu (*service*) a používaní *twig extension*.

Bylo zapotřebí opět vytvořit entitu jako v předchozím projektu. V mém případě *Greeting*, která nese pouze atribut *id* a *greeting* respektive pozdrav. Entita nám opět poslouží ve zjednodušení jako kontejner na data při komunikaci s databází, kde jsem připravil tabulku s jednotlivými pozdravy.

Dalším krokem bylo vytvořit repozitář, kde jsem naimplementoval metodu pro vrácení náhodného pozdravu. Při řešení jsem narazil na problém, kdy Doctrine ORM neposkytuje možnost navrácení náhodného řádku/ záznamu z databáze. Je zde ovšem mnoho způsobů, jak danou situaci vyřešit. Zvolil jsem přidání přizpůsobené číselné funkce. Proto bylo zapotřebí přidat do projektu novou třídu s *random* funkcí viz Výpis 11.

```
1.  namespace AppBundle\DQL;
2.
3.  use Doctrine\ORM\Query\Lexer;
4.  use Doctrine\ORM\Query\Parser;
5.  use Doctrine\ORM\Query\SqlWalker;
6.  use Doctrine\ORM\Query\AST\Functions\FunctionNode;
7.
8.  class RandFunction extends FunctionNode
9.  {
10.     ...
11.
12.     public function parse(Parser $parser)
13.     {
14.         $parser->match(Lexer::T_IDENTIFIER);
15.         $parser->match(Lexer::T_OPEN_PARENTHESIS);
16.         $parser->match(Lexer::T_CLOSE_PARENTHESIS);
17.     }
18.
19.     /**
20.      * @param SqlWalker $sqlWalker
21.      * @return string
22.      */
23.     public function getSql(SqlWalker $sqlWalker)
24.     {
25.         return 'RAND()';
26.     }
27. }
```

Výpis 11: Implementace *RandFunction*

Po vytvoření *RandFunction*, bylo důležitě zaregistrovat tuto třídu v souboru *config.yml* daného projektu viz Výpis 12.

```
...
dql:
    numeric_functions:
        Rand: AppBundle\DQL\RandFunction
...
```

Výpis 12: Zaregistrování *RandFunction* v *config.yml*

Zaregistrováním bylo možné funkci *RAND()* používat přímo v dotazech v podobě *…addSelect('RAND() as HIDDEN rand')->orderBy('rand')*. Stačilo tedy v našem repozitáři, za pomocí dotazu Doctrine, použít předchozí funkci k vyselektování náhodného záznamu, respektive pozdravu viz Výpis 13.

```
...
use Doctrine\ORM\EntityRepository as BaseRepository;

class GreetingRepository extends BaseRepository
{
    ...
    public function getRandomEntity()
    {
        return  $this->createQueryBuilder('g')
            ->select('g.greeting')
            ->addSelect('RAND() as HIDDEN rand')
            ->addOrderBy('rand')
            ->setMaxResults(1)
            ->getQuery()
            ->getOneOrNullResult();
    }
}
```

Výpis 13:Ukázka repozitáře s metodou pro náhodný pozdrav

Poslední částí bylo naimplementovat *ExtensionClass*. Twig Extension [15] se používá v případě, kdy potřebujeme vytvořit různé Twig funkce, filtry, testy a další. V mém případě se jedná o funkci, proto bylo důležité definovat *getFunctions()*, ve které se odkazuji na funkci *getRandomGreeting*, která je zde také definovaná viz Výpis 14. Funkce *getRandomGreeting* nedělá nic jiného, než že nám vrátí náhodný pozdrav pomocí nadefinované metody *getRandomEntity()* v našem repozitáři.

```
class GreetingExtension extends \Twig_Extension
{
    ...

    /**
     * @return array
     */
    public function getFunctions()
    {
        return array(
            new \Twig_SimpleFunction('greeting', array($this, 'getRandomGreeting')),

        );
    }
```

```
14.
15.     /**
16.      * @return string
17.      * @throws \Doctrine\ORM\NonUniqueResultException
18.      * @throws \Doctrine\ORM\NoResultException
19.      */
20.     public function getRandomGreeting()
21.     {
22.         return current($this->doctrine->getRepository(Greeting::class)-
    >getRandomEntity());
23.     }
24.     ...
25. }
```

Výpis 14: Ukázka *GreetingExtension*

Přidání náhodného pozdravu do potřebné Twig šablony se provede vyvoláním GreetingExtension pomocí definování *{{ greeting() }}*.

4.4.2 Omezení na 15 minut přihlášení

Úkolem bylo vytvořit službu pro ošetření maximální doby přihlášení při neaktivitě. V našem případě to znamenalo, že pokud je přihlášený uživatel neaktivní déle jak 15 minut, provede se automaticky odhlášení uživatele.

K vyřešení úkolu jsem se rozhodl provést implementaci s *kernel listener*. Vytvořil jsem nový parametr v souboru *parameters.yml.dist* v */configu* projektu pro případ budoucí změny doby neaktivity. Parametr *session_max_idle_time: 900* nese limit maximální doby neaktivity, kde hodnota 900 je počet sekund (v našem případě 15 minut: 15*60 sekund).

Hlavním krokem tohoto úkolu bylo vytvoření nové třídy *SessionIdleHandler*, která v konstruktoru předává následující parametry:

- *SessionInterface $session* – objekt, který nám umožňuje ukládat informace o uživatelích mezi jednotlivými požadavky. V našem případě poslouží pro získání neaktivity uživatele.
- *TokenStorageInterface $securityToken* – objekt nesoucí informace o ověření a zabezpečení. V našem případě přihlášení uživatele.
- *RouterInterface $router* – je interface, který slouží k přesměrování.
- *$expiredTime = 0* – parametr s maximální možnou neaktivitou. V našem případě vytvořený parametr *session_max_idle_time.*

Další metodou je *onKernelRequest* viz Výpis 15. Zde řeším kontrolu neaktivity a případné odhlášení. Tělo metody se skládá ze čtyř hlavních *if* podmínek.

- První podmínkou otestujeme, zda je *token $securityToken* roven *null* hodnotě. Tedy jestli existuje nějaké ověření v podobě přihlášení. V případě pravdivosti podmínky se již nepokračuje dál ve funkci viz Výpis 15.
- Druhá podmínka je řešena v případě nesplnění první. To znamená, že *token* existuje, a proto ověříme, zda se jedná o uživatele. Pomocí funkce *!is_object()* a její následné negace otestujeme, zda se jedná, respektive nejedná o objekt. V případě splnění podmínky (anonymní ověření), tedy že se nejedná o objekt, se již nepokračuje dále ve funkci viz Výpis 15.

- Třetí podmínka nám v případě nesplnění dvou předešlých ověří, zda se jedná o *MASTER_REQUEST*. Tedy o požadavek, který přišel od původního uživatele. Druhým typem požadavku je ten, který je prováděn uvnitř: *SUB_REQUEST*. V tomto případě dojde ke splnění podmínky a opět k ukončení průchodu funkcí *onKernelRequest* viz Výpis 15.
- Čtvrtá podmínka řeší hlavní problematiku. Ta nastane v případě nesplnění třech předešlých, kde jsme ověřili, zda se jedná o přihlášeného uživatele, který provádí požadavek (request). Symfony obsahuje komponentu *HttpFoundation* [16], která mimo jiné obsahuje dobrý subsystém pro relace (*session*). Proto po ověření, zda *$this->expiredTime* je větší než 0, spustíme relaci pomocí metody *start()*. Ze *session* následně dostaneme pomocí *$this->session->getMetadataBag()->getLastUsed()* čas v sekundách, kdy uživatel projevil naposledy nějakou aktivitu. Získaný čas představuje počet sekund uplynulých od 1.1.1970. Pro získání celkové délky neaktivity, získanou hodnotu odečteme od aktuálního počtu sekund funkce *time()* a následně ověříme v *if* podmínce, zda byla poslední aktivita před více než 900 sekundami. V případě splnění podmínky provedeme odhlášení uživatele nastavením *$this->securityToken* na hodnotu *null*, vypíšeme hlášku uživateli, a provedeme přesměrování na kontroler s přihlášením viz Výpis 15.

```
1.  class SessionListener
2.  {
3.      ...
4.
5.      public function onKernelRequest(GetResponseEvent $event)
6.      {
7.          if (null === $token = $this->securityToken->getToken()) {
8.              return;
9.          }
10.         if (!is_object($user = $token->getUser())) {
11.             // e.g. anonymous authentication
12.             return;
13.         }
14.         if (HttpKernelInterface::MASTER_REQUEST !== $event->getRequestType()) {
15.             return;
16.         }
17.         if ($this->expiredTime > 0) {
18.             $this->session->start();
19.             $time = time() - $this->session->getMetadataBag()->getLastUsed();
20.             if ($time > $this->expiredTime) {
21.                 $this->securityToken->setToken(null);
22.                 $this->session->getFlashBag()-
    >set('info', 'You have been logged out due to inactivity.');
23.                 $event->setResponse(new RedirectResponse($this->router-
    >generate('fos_user_security_login')));
24.             }
25.         }
26.     }
27. }
```

Výpis 15: Ukázka řešení *SessionListener*, který kontroluje neaktivitu

Nakonec jsem zaregistroval daný *listener* jako službu (*service*) v souboru *services.yml*, kde jsem mimo jiné definoval jednotlivé parametry metody: *@session*, *@security.token_storage*, *@router* včetně vytvořeného parametru pro nastavení limitu: *%session_max_idle_time%*.

4.4.3 Vytvoření administrace

Další částí bylo podílet se na administraci. Konkrétně se jednalo o sekci můj profil a vytvoření stránky. Zadáním bylo vytvořit výpis uživatelů a potřebné akce s nimi. Při implementaci jsem použil již získané zkušenosti z předešlých projektů.

K práci s daty a smysluplnému vypisování do tabulek jsem použil knihovnu třetí strany SgDatatablesBundle [17]. Jedná se o balíček, který integruje do Symfony plugin jQuery DataTables, který uspořádá získaná data do datové tabulky, dále jen DataTable. Balíček také mimo jiné pracuje s již zmíněnou Doctrine ORM. Velkou výhodou DataTable je, že umožňuje mimo jiné mnoho možností, jak uspořádat data, nastavení jednotlivých sloupců, pokročilé filtrování dat, akce se záznamy a dotazy na databázi pomocí Doctrine ORM apod.

Protože byl projekt na začátku, bylo zapotřebí zde balíček SgDatatablesBundle nainstalovat. Po stažení balíčku za pomocí composeru je nutné ho přidat mezi stávající balíčky v souboru *app/AppKernel.php*, kde ho zaregistrujeme pomocí *new Sg\DatatablesBundle\SgDatatablesBundle()* a přidat konfiguraci do *routing.yml*. Protože balíček požívá celou řadu CSS stylů a JavaScriptů, je nutné pro následující použití a správnou funkčnost všechny náležitosti ve správném pořadí zaimplementovat do základní twig šablony.

K samotnému vytvoření DataTable je zapotřebí vytvoření Entity s potřebnými hodnotami pro následné spojení s databází a práci s daty jako v předešlém projektu. Dále je nutné vytvořit třídu, v mém případě *UserDatatable*, která rozšiřuje *AbstractDatatable* viz Výpis 16. Jednou z hlavních funkcí této třídy je funkce *buildDatatable*, kde implementuji jednotlivá nastavení proměnných pro danou DataTable:

- *$this->options* - v této hodnotě typu *array* jsem nastavil veškeré potřebné možnosti týkající se především vizuální stránky Datatable. Jedná se o nastavení tříd, které definují styly Bootstrapu pro danou tabulku. Dále povolení políčka pro filtrování a jeho umístění v hlavičce tabulky a další nastavení, jako řazení buněk viz Výpis 16.
- *$this->language* - tato proměnná se stará o internalizaci, tedy jazyk.
- *$this->columnBuilder* – je proměnná typu *array*, definující jednotlivé sloupce tabulky. Zde jsem naimplementoval veškeré sloupce. Postup přidávání sloupců je obdobný jako při vytváření tříd pro formuláře (*form TYPE*). Jako první definuji název a typ sloupce: *add('username', Column::class.* SgDatatables balíček obsahuje několik tříd s typy sloupců, jako jsou například třídy: *Column* (reprezentuje základní sloupec), *DateTimeColumn* (reprezentuje sloupec pro datum a čas), *ActionColumn* (představuje sloupec pro potřebné akce nad záznamy) a další. U každého sloupce jsem dále definoval základní možnosti, jako jsou: titulek (*'title'*), možnost vyhledávání (*'searchable' => true*), řazení a další, dle požadavků v zadání. Posledním sloupcem tabulky je sloupec s tlačítky pro jednotlivé akce se záznamy. Pro tento typ sloupce je zde třída *ActionColumn*, kde opět definuji jednotlivé nastavení buněk a dále pak jednotlivé možnosti pro danou akci v podobě cesty akce (*'route'*), předaných parametrů (*'id' => 'id'*), název tlačítka (*'label'*), stylů a atributů tlačítka, ověření superadmina, titulek a další viz Výpis 16.

```
...
class UserDatatable extends AbstractDatatable
{
    ...
    public function buildDatatable(array $options = array())
    {
        $this->options->set(array(
            'classes' => Style::BOOTSTRAP_3_STYLE,
            'stripe_classes' => [ 'strip1', 'strip2', 'strip3' ],
            'individual_filtering' => true,
            'individual_filtering_position' => 'head',
            'order' => array(array(0, 'asc')),
            'order_cells_top' => true,
            'search_in_non_visible_columns' => true,
        ));

        $this->language->set( [
            'cdn_language_by_locale' => true,
            'language_by_locale' => true,
            'language' => 'cs'
        ] );

        $this->columnBuilder
            ->add('username', Column::class, array(
                'title' => $this->translator->trans( 'name', [], 'datatables' ),
                'searchable' => true,
                'orderable' => true,
                'filter' => array(TextFilter::class, array(
               )),
            ))
            ...
            ->add(null, ActionColumn::class, array(
                'title' => $this->translator-
>trans( 'actions', [], 'datatables' ),
                'start_html' => '<div class="start_actions">',
                'end_html' => '</div>',
                'actions' => array(
                    array (
                        'route' => 'app_admin_users_superadminedituser',
                        'label' => $this->translator-
>trans( 'edit', [], 'datatables' ),
                        'route_parameters' => array (
                            'id' => 'id',
                        ),
                        'render_if' => function()
                        {
                            return $this->authorizationChecker-
>isGranted( User::ROLE_SUPER_ADMIN );
                        },
                        'attributes' => array (
                            'rel' => 'tooltip',
                            'title' => $this->translator-
>trans( 'edit', [], 'datatables' ),
                            'class' => 'datatable-link btn btn-primary btn-xs',
                            'role' => 'button'
                        ),
                    ), ...
                )
            ));
    }...
}
```

Výpis 16: Ukázka vytvoření, nastavení a přidání sloupců ve třídě *UserDatatable*

Dále jsem vytvořil *ProfileType*, který je typu *entity Field Type* [18]. Jde o třídu definující formulář (pole), které je navrženo pro načtení možností z Doctrine entity. Toto pole pak využívám k vytváření nového uživatele, nebo k vypsání hodnot z databáze při akcích editace.

Příprava kontroleru *UserController* byla poslední částí user administrace. Hlavní funkcí je *indexAction*, která se stará o načtení dat z databáze a následné uspořádání do již připravené DataTable. Pro použití naší třídy *UserDatatable* v kontroleru bylo zapotřebí ji zaregistrovat v souboru *services.yml*. Nyní jsem pomocí funkce *get('app.datatable.user')* načetl zaregistrovanou *UserDatatable* a následně ji pomocí funkce *buildDatatable()* vytvořil a předal ji do twig šablony viz Výpis 17. O naplnění tabulky daty z databáze se stará balíček SgDatatableBundle spolu s Doctrine OMR.

```
...
public function indexAction(Request $request)
    {
        $isAjax = $request->isXmlHttpRequest();
        $datatable = $this->get('app.datatable.user');
        $datatable->buildDatatable();

        if ($isAjax) {
            $responseService = $this->get('sg_datatables.response');
            $responseService->setDatatable($datatable);
            $responseService->getDatatableQueryBuilder();
            return $responseService->getResponse();
        }

        return $this->render('admin/user/index.html.twig', array(
            'datatable' => $datatable,
        ));
    }
...
```

Výpis 17: Ukázka vytvoření *UserDatatable* ve funkci *indexAction()*

Dalšími akcemi kontroleru byly akce pro tlačítka, které jsem implementoval v DataTable. Jedná se o akce nad jednotlivými záznamy, jako jsou editace, mazání apod. Protože jsem podobný úkol již řešil, nebylo obtížné vše připravit. Práci s daty a akce nad nimi mi opět ulehčil framework Doctrine ORM, kde funkce pro mazání a editaci byly obdobné jako v případě projektu Wave. Nakonec jsem připravil soubory pro překlady.

Po úspěšném dokončení user administrace jsem dostal podobný úkol. Jednalo se o rozšíření administrace pro správu stránek. Zadáním bylo vytvořit výpis stránek v DataTable a následné akce s nimi, jako jsou: úpravy, mazání, vytváření apod. Pro splnění úkolu jsem využil zkušeností z předchozího úkolu, ve kterém byla implementace podobná.

4.5 Práce na interním projektu PageChecker

V průběhu mé bakalářské praxe jsem pracoval na projektu PageChecker od jeho založení. Má práce obsahovala hned několik dílčích úkolů na modulech Pixel Checker a Api Checker, které tvoří hlavní náplň mé odborné praxe.

4.5.1 Modul Pixel Checker

Z počátku jsem pracoval na modulu Pixel Checker, který, jak jsem již zmínil, se bude starat o kontrolu přítomnosti pixelů na daných stránkách. Protože se jedná o začínající projekt, bylo důležité doinstalovat a zaregistrovat v *AppKernel.php* potřebné balíčky, jako jsou DoctrineMigrationsBundle, SgDatatablesBundle, FOSUser a podobně.

Následně jsem přichystal administraci s potřebnými akcemi včetně přehledu pixelů v tabulce. Při implementaci jsem opět použil balíček SgDatatablesBundle. K vyřešení jsem tedy využil již získané zkušenosti s touto problematikou, protože se jednalo o podobný postup jako v předešlém projektu. Řešením tedy byla příprava entity *PixelFire* s veškerými *properties* včetně setterů a getterů. Dále připravení entity *Field Type* pro formulář vytvoření a další akce se záznamy. Následně jsem naimplementoval *PixelDatatable* a *PixelController*, podobně jako v předešlém projektu. Poslední částí tohoto úkolu byla příprava jednotlivých Twig šablon.

Dalším úkolem na modulu Pixel Checker bylo připravit seznam s veškerými URL adresami, na nichž se vyskytuje přítomnost pixelů, ke kontrole a sledování. Úkolem bylo projít desítky projektů společnosti Webvalley, ve kterých jsem hledal přítomnost pixelů v šablonách. Tento úkol nebyl složitý, avšak časově náročný. Důvodem časové náročnosti bylo seznámit se s projekty a pochopit jejich strukturu a následně v nich nalézt šablony s výskytem pixelů neboli kusu kódu ke kontrole. Ve finální části se jednalo o 180 URL adres.

4.5.2 Modul Api Checker

Poslední úkoly byly na modulu Api Checker. Jedná se o modul, který se bude starat o kontrolu webů a aplikací pomocí jejich API (aplikační rozhraní).

Jedním z hlavních úkolů bylo připravit službu pro kontrolu textového řetězce JSON doručeného z API daného kontrolovaného projektu nebo aplikace. Na tento úkol postupem času navazovaly další menší úkoly v podobě připravení pravidel ke kontrole, připravení administrace, a další úpravy spočívající ve vytvoření přihlašovací stránky a front-end úprav.

Prvním krokem bylo vytvořit potřebné služby, modely a entity včetně getterů a setterů pro *properties*:

- **Rule** – entita, která obsahuje *properties*: *$id*, *$apiCheck* (propojení s danou ApiCheck entitou), *$path* (cesta ke kontrolované hodnotě v JSON řetězci), *$rule* (číslo pravidla, kterým se má kontrolovat hodnota), *$ruleParam* (potřebné parametry k pravidlům), *$message* (obsahuje chybovou zprávu, která bude uložena do historie v případě chyby), *$okMessage* (obsahuje úspěšnou zprávu). Dále jsem zde definoval konstanty (např: *const BIGGER = 2;* pro porovnání, zda je hodnota větší), které obsahují identifikační číslo pravidla, jejichž využití uplatním ve službě *RuleChecker* při průchodu konstrukcí *switch*.
- **RulesChecker** – je služba starající se o porovnávání dle nastavených pravidel. Hlavní částí služby je funkce *evaluate()*, kde jsem definoval veškerá pravidla, která byla zadána. Vstupními parametry funkce jsou: entita *Rule*, $*oldValue* (stará hodnota) a *$newValue* (nová hodnota). Tělem funkce je konstrukce *switch* viz Výpis 18. Konstrukci tvoří celkem 21 pravidel, které řeší porovnávání staré hodnoty/ datumu s novou hodnotou/ datumem (<, >, <=, >=, ===, *!==*, *between* apod.). Dále pak rozšířené porovnávání pomocí parametrů uložených v atributu *ruleParam* daného záznamu *Rule*, jako jsou například „je starší/ mladší o

x hodin“ viz Výpis 18, hledání slova v řetězci string apod. Dané pravidlo se zvolí při průchodu konstrukcí dle nastaveného čísla v atributu *rule* entity *Rule,* kde jsem definoval již zmíněné konstanty. Návratovým typem každého *case* je *boolean*. V ukázce můžeme vidět pravidlo, zda je datum mladší o *X* hodin, kde číslo *X* je uložené v JSON řetězci *ruleParam*, konkrétně parametr s klíčem *hours*.

```
1.  class RulesChecker
2.  {
3.      public function evaluate(Rule $rule, $oldValue, $newValue)
4.      {
5.          switch($rule->getRule())
6.          {
7.              ...
8.              case Rule::YOUNGER_X_HOURS:
9.                  $date = new DateTime($oldValue);
10.                 $date->add(new DateInterval('PT'.$rule-
    >getRuleParam()['hours'].'H'));
11.                 $oldValue = $date->format('Y-m-d H:i:s');
12.                 return $newValue >= $oldValue;
13.             ...
14.         }
15.     }
```

Výpis 18: Ukázka pravidla ověření stáří data pomocí parametru a funkce *DateInterval()*

Druhou implementovanou funkcí služby *RuleChecker* je *normalizeValue()*, která se stará pouze o to, aby v případě, že vstupní hodnota *$value* je pole, se navrátila hodnota *$value[0]*, tedy hodnota na první pozici.

- **ApiCheck** – entita, která zastupuje jednotlivé kontroly, uchovává cestu (*$url*) k API projektu, které navrátí JSON. Dále obsahuje *properties*: *$id*, *$method*, *$service* (nastavení služby pro provedení kontroly, kde tato služba bude vždy rozšiřovat službu *BaseApiService* např. o nastavení budoucí kontroly), *$serviceParams* (parametr pro službu), *$rules* (propojení s Rule entitou), *$lastApiResponse* (propojení s LastApiResponse entitou), *$lastCheck* (datum poslední kontroly), *$nextCheckAfter* (datum budoucí kontroly).
- **ApiResponse –** model, který bude sloužit k uchování získaných dat z API kontrolovaného projektu. Model bude uchovávat textový řetězec JSON (*$body*), stavový kód, zda byl požadavek úspěšně zpracován (*$statusCode*), a čas provedení požadavku (*$responseTime*).
- **LastApiResponse** – entita, sloužící pro uchování dat z proběhlých kontrol daného *ApiChecku*. Tato entita obsahuje *properties*: *$id*, *$apiCheck* (propojení s danou *ApiCheck* entitou), *$timestamp* (časové razítko kontroly), *$service* (služba kontroly) a *$body* (kontrolované hodnoty). Záznamy nám poslouží pro získání naposledy kontrolované hodnoty daného A*piChecku*, kterou využijeme při kontrolování s novou získanou hodnotou.
- **ApiCheckProblemHistory –** je entita rozšiřující entitu *ProblemHistory*. Jedná se o logiku, která zde již byla implementovaná a řeší problematiku zaznamenávání historie s chybami pro následné ukládání do databáze. Tuto entitu jsem rozšířil o funkce *createInfo()*, *createWarning()* a *createError()*, kde vstupními parametry jsou: *ApiCheck* a *message* (zpráva specifikující kontrolu). Další rozšiřující funkcí, kterou vyvolám za pomocí *self* v těle třech předešlých, je funkce *create()*, která vytvoří potřebný záznam (*ProblemHistory*) pro budoucí řešení notifikací a zaznamenávání chyb viz Výpis 19. Vstupními parametry jsou: předaný

ApiCheck, definovaný *level* a předaná *message*. Využití tohoto rozšíření blíže popíši níže při implementaci funkce *handleData()*, která je součásti služby *BaseApiService* viz Výpis 21.

```
1.  class ApiCheckProblemHistory extends ProblemHistory
2.  {
3.  ...
4.      public static function createInfo(ApiCheck $apiCheck, $message)
5.      {
6.          return self::create($apiCheck, ProblemHistory::LEVEL_INFO, $message);
7.      }
8.
9.      public static function createWarning(ApiCheck $apiCheck, $message)
10.     {
11.         return self::create($apiCheck, ProblemHistory::LEVEL_WARNING, $message);
12.     }
13.
14.     public static function createError(ApiCheck $apiCheck, $message)
15.     {
16.         return self::create($apiCheck, ProblemHistory::LEVEL_ERROR, $message);
17.     }
18.
19.     public static function create(ApiCheck $apiCheck, $level, $message)
20.     {
21.         $problemHistory = new self();
22.
23.         $problemHistory->setLevel($level);
24.         $problemHistory->setApiCheck($apiCheck);
25.         $problemHistory->setProblemDescription($message);
26.
27.         return $problemHistory;
28.     }
29. ...
30. }
```

Výpis 19: Ukázka rozšíření třídy *ProblemHistory* o potřebné vytvoření záznamu.

V následujícím kroku jsem pracoval na implementaci samotné služby *BaseApiService*, která se bude starat o získání JSON řetězce z API aplikovaného na kontrolovaném projektu. Poté tento řetězec zpracovat a nalézt v něm požadovanou hodnotu k ověření, na které následně provedeme kontrolu dle stanovených pravidel. Nakonec uložit kontrolu a případné chyby do databáze.

Pro získání dat z API jsem naimplementoval funkci *fetchData()*, která je součástí služby *BaseApiService.* Vstupním parametrem funkce je entita *ApiCheck*, tedy záznam z databáze, který nám specifikuje danou kontrolu viz Výpis 20. Pro získání JSON řetězce z URL adresy daného *ApiChecku*, která směřuje na dané API v kontrolovaném projektu, jsem použil knihovnu *cURL* [19]. Toto rozšíření umožňuje vytvářet požadavky na server pomocí mnoha protokolů (pro nás důležité *http* a *https*). Jako první tedy provedu inicializaci relace *cURL* pomocí funkce *curl_init()*. Následně pomocí funkce *curl_setopt()*, která poskytuje různá nastavení, a pomocí její možnosti *CURLOPT_RETURNTRANSFER* převedu návratovou hodnotu na textový řetězec JSON (datový typ *string*). V dalším kroku uložím *string* (*body*), čas požadavku (*responseTime*) a stavový kód (*statusCode*) do modelu *ApiResponse*, který je návratovou hodnotou funkce *fetchData(),* a ukončím relaci *cURL* pomocí *curl_close()* viz Výpis 20.

```
...
protected function fetchData(ApiCheck $apiCheck)
    {
        $ch = curl_init($apiCheck->getUrl());

        curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);

        $timeStart = microtime(true);
        $response = curl_exec($ch);
        $timeEnd = microtime(true);

        $responseObject = new ApiResponse();

        $responseObject->body = $response;
        $responseObject->statusCode = curl_getinfo($ch,CURLINFO_HTTP_CODE);
        $responseObject->responseTime = $timeEnd - $timeStart;

        curl_close($ch);

        return $responseObject;
    }
...
```

Výpis 20:Ukazká funkce *fetchData()* služby *BaseApiService*

Druhou implementovanou funkcí služby *BaseApiService* je metoda *handleData()*, která řeší hlavní logiku kontroly a následného ukládání problémů. Vstupními parametry je daný *ApiCheck*, *NotificationCollector* (slouží k zaznamenávání chyb pro následné uložení do databáze a budoucí řešení notifikací) a model *ApiResponse*, který jsme získali pomocí předchozí implementované funkce *fetchData()*.

V prvním kroku funkce *handleData()* si uložím do proměnné *$jsonData* aktuální řetězec JSON. To provedu za pomocí funkce *parseJsonBody()*, která je součástí našeho modelu, v jejíž definici konvertuji získané data za pomocí *json_decode()* do PHP proměnné. Následně získám poslední provedenou kontrolu daného *ApiChecku*, která je uložená v tabulce *LastApiResponse*, a uložím ji do proměnné *$lastCheck* (v případě že neexistuje, navrátím *null*). V dalším kroku ověřím v hlavní podmínce *if,*, zda jsou k dispozici obě proměnné (*$lastCheck* a *$jsonData*) viz Výpis 21.

V případě nesplnění podmínky se kontrola neprovede a přidá se notifikace za pomocí již připravené logiky *NotificationCollector* třídy. Zde se uchovávají jednotlivé notifikace v poli *array* pro následné zpracování. Dále je zde implementovaná funkce *addNotification()*, pro přidání nové notifikace do již zmíněného pole. Vstupním parametrem je entita *ProblemHistory*, kterou vytvoříme za pomocí našeho již zmíněného rozšíření *ApiCheckProblemHistory* (rozšíření entity *ProblemHistori*) a požadované funkce viz Výpis 21. V tomto případě pomocí implementované funkce *createError()* vytvoříme požadovaný záznam. Při vytváření záznamu se v těle metody vyvolá *self* funkce *create()* viz Výpis 19, která provede vytvoření záznamu v podobě vytvoření nového objektu *problemHistory*. Následně do tohoto objektu uložíme daný *apiCheck*, požadovanou zprávu a nastavíme *level* na *LEVEL_ERROR*, který je definovaný v entitě *ProblemHistory*.

V případě splnění podmínky si uložím do proměnných nový a starý řetězec JSON získaný z API. Následně pomocí cyklu *foreach* procházím veškeré nastavené kontroly pro daný *ApiCheck*, kde si při každém průchodu uložím do proměnných *$oldValue* a *$newValue* starou a novou hodnotu ke kontrole. Danou hodnotu získám ze starého/ nového řetězce za pomocí cesty k dané hodnotě, která je součástí

každé kontroly (entita *Rule* proměnná *$path*) viz Výpis 21. Při získávání hodnoty použiji již zmíněnou funkci *normalizeValue()* pro případ, že by se jednalo o pole. Následně provedu ověření za pomocí již představené funkce *evaluate()*, která mi určí, zda je kontrola podle nastaveného pravidla korektní, či nikoliv. V případě splnění pravidla (návratová hodnota *true*) se přidá notifikace, podobně jako v předešlé situaci při nesplnění podpínky *if*, pomocí rozšířené funkce *createInfo()*, ve které nastavíme proměnnou *level* na hodnotu *LEVEL_INFO*. Při neúspěšné kontrole (návratová hodnota *false*) vytvoříme notifikaci pomocí rozšířené funkce *createWarning()* s hodnotou *level = LEVEL_WARNING*.

Veškeré notifikace s chybami jsou nahrané do databáze po zavolání funkce *checkComplete()* viz Výpis 21, která se stará o nahrání problémových záznamů. Funkce prochází jednoduchým cyklem *foreach* pole s notifikacemi a hledá pouze ty, které mají nastavený *level* na hodnotu *LEVEL_WARNING* nebo *LEVEL_ERROR*. Tyto notifikace následně uložíme do databáze, konkrétně do tabulky *ProblemHistory*.

```
1.  ...
2.  public function handleData(ApiCheck $apiCheck, NotificationCollector $notificationRe
    porter, ApiResponse $data)
3.      {
4.          $jsonData = $data->parseJsonBody();
5.          /** @var LastApiResponse $lastCheck */
6.          $lastCheck = $apiCheck->findLastCheck();
7.
8.          if($jsonData && $lastCheck){
9.              $oldDataJsonExploer = new JsonStore($lastCheck->getBody());
10.             $newDataJsonExploer = new JsonStore($jsonData);
11.
12.             $rulesChecker = new RulesChecker();
13.
14.             /** @var Rule $rule */
15.             foreach($apiCheck->getRules() as $rule){
16.                 $oldValue = RulesChecker::normalizeValue($oldDataJsonExploer-
    >get($rule->getPath()));
17.                 $newValue = RulesChecker::normalizeValue($newDataJsonExploer-
    >get($rule->getPath()));
18.
19.                 if($rulesChecker->evaluate($rule, $oldValue, $newValue)){
20.                     $notificationReporter-
    >addNotification(ApiCheckProblemHistory::createInfo($apiCheck, $rule-
    >getOkMessage()));
21.                 }else{
22.                     $notificationReporter-
    >addNotification(ApiCheckProblemHistory::createError($apiCheck, $rule-
    >getMessage() . " ({$oldValue} -> {$newValue})"));
23.                 }
24.             }
25.         }else{
26.             $notificationReporter-
    >addNotification(ApiCheckProblemHistory::createError($apiCheck, "Can't parse request
     body!"));
27.         }
28.
29.         $notificationReporter->checkComplete();
30.     }
31. ...
```

Výpis 21: Ukázka funkce *handleData()*

V návaznosti na dvě předešlé funkce (*fetchData()* a *handleData()*), jsem implementoval hlavní funkci *makeCheck()*, která bude inicializována a stará se o kompletní logiku služby *BaseApiService*. V těle funkce zároveň řeším vytvoření záznamu v tabulce *LastApiResponse*, který bude uchovávat data o proběhlé kontrole. Průchodem funkce nastavím veškeré potřebné *properties* entity *LastApiResponse* včetně inicializace již definovaných funkcí. Tedy nejdříve získání modelu pomocí funkce *fetchData()*, který následně předám funkci *handleData()* pro provedení kontroly a uložení chyb. Dále zde řeším nastavení budoucí kontroly, jedná-li se o spuštění *cornem* (*TRIGGER_CORN*), a nastavení data poslední kontroly pro daný *apiCheck.* Nakonec vše uložím do databáze pomocí funkce *persist()* a *flush*(), jako v předešlých projektech viz Výpis 22.

```
1.  public function makeCheck(ApiCheck $apiCheck, NotificationCollector $reporter, $trig
    gered = LastApiResponse::TRIGGER_CRON)
2.      {
3.          $entityManager = $this->doctrine->getManager();
4.          $log = new LastApiResponse();
5.          $log->setTriggered($triggered);
6.          $log->setService($apiCheck->getService());
7.          $log->setApiCheck($apiCheck);
8.
9.          $data = $this->fetchData($apiCheck);
10.         $log->setBody($data->body);
11.
12.         $this->handleData($apiCheck, $reporter, $data);
13.
14.         if($triggered === LastApiResponse::TRIGGER_CRON){
15.             $this->setNextRunAfter($apiCheck);
16.         }
17.
18.         $apiCheck->setLastCheck(new \DateTime());
19.
20.         $entityManager->persist($log);
21.         $entityManager->flush();
22.     }
23. ...
```

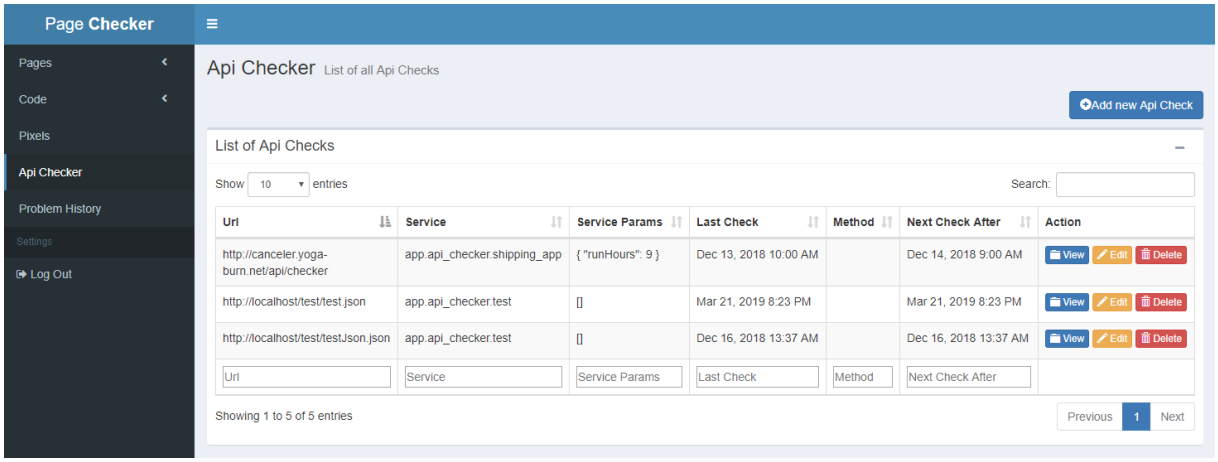
Výpis 22: Ukázka funkce *makeCheck()*

Následně jsem připravil inicializaci, pro možnost spuštění kontroly pomocí příkazového řádku. Symfony framework poskytuje mnoho příkazů s použitím scriptu *bin/console*, které jsou tvořeny pomocí komponenty Console. Pro vyřešení jsem definoval třídu *ApiCheckerCommand*, která je rozšířením třídy *Command*, v jejímž těle jsem definoval název a funkci *execute()* pro inicializaci.

Ve funkci *execute()* definuji veškeré napojení s databází v podobě získání všech *Apicheck* kontrol, které se mají provést. Tedy vyselektuji záznamy, jejichž atribut *nextCheckAfter* je staršího data než aktuální datum. Následně pomocí cyklu *foreach* projdu veškeré získané *apiCheck* záznamy a pro každý provedu kontrolu pomocí služby *BaseApiService* (funkce *makeCheck()*).

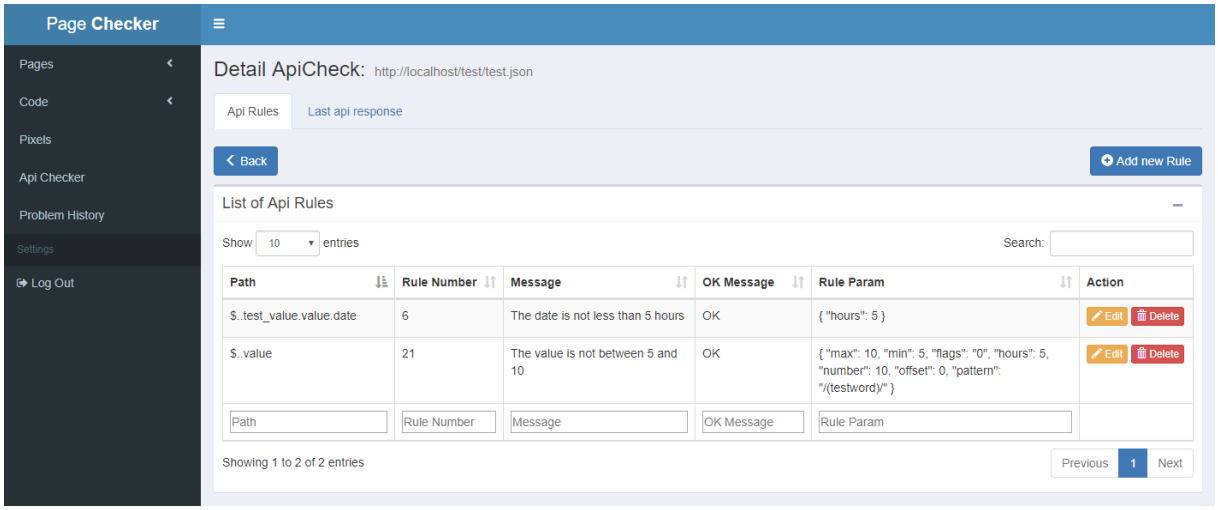
Další fází bylo testování služby. Vytvořil jsem si vlastní testovací JSON řetězec, který simuloval API, a průběžně jsem v něm měnil kontrolované hodnoty. Dále jsem připravil záznam *apiCheck* a *rule*, kde jsem měnil jednotlivá aplikovaná pravidla a kontroloval korektnost provedení a případné ukládání informací do databáze. Kontroly jsem spouštěl pomocí příkazu *php bin/console app:api-checker* definovaného v třídě *ApiCheckerCommand*.

Následovala příprava administrace. Vytvořil jsem hlavní přehled všech záznamů *ApiCheck* s požadovanými akcemi: editace, mazání, podrobnosti a možnost vytvoření nového záznamu viz Obrázek 2. Akce podrobnosti zobrazovala detailní informace o daném *apiChecku*, který obsahoval tabulky s nastavenými pravidly (včetně možných akcí: úpravy, mazání a vytvoření) a historii provedených kontrol pro daný *apiCheck* (tabulka *LastApiResponse*). Protože jsem se s tímto úkolem již několikrát setkal, nebylo obtížné jeho vyřešení. K aplikování jsem použil již popsané metody a balíčky (sgDataTableBundle) a případně je rozšířil dle požadavků. Jednalo se například o vyřešení vypsání objektu do čitelné formy (v případě sloupců *ServiceParams* a *ruleParam*). K vyřešení jsem použil funkci *getLineFormatter()*, která je součástí balíčku sgDataTableBundle a umožňuje nám formátování jednotlivých sloupců. Zde jsem naformátoval požadovaný sloupec tabulky za pomocí funkce *json_encode($row['serviceParams'], JSON_PRETTY_PRINT)* do čitelné podoby.



Obrázek 2: Náhled na vytvořenou administraci s Api Check záznamy

Dalším problémem bylo vykreslení zmíněných DataTable s historií (tabulka *LastApiResponse*) a pravidly (tabulka *Rule*) na kartě podrobněji daného *ApiChecku*. Důvodem bylo nepodporování vytvoření více DataTables v jedné funkci kontroleru (v případě definování druhé se již nenaplní daty). Problém jsem vyřešil implementací twig šablony, ve které definuji přepínací *taby* viz Obrázek 3. Tuto šablonu následně rozšířím druhou šablonu, do které posílám DataTable k vykreslení. V dalším kroku vytvořím v *ApiCheckControlleru* funkce pro jednotlivé sestavení DataTable, jako v předchozích případech, kde pro návratový *render* nastavím rozšiřující šablonu. Následně jsem nastavil tlačítkám, tvořící *taby*, požadovanou cestu k odpovídající funkci v kontroleru, pro vytvoření dané tabulky, a také zajistil předání *id* daného *ApiChecku* do kontroleru jako atribut parametr viz příklad: *href="{{ path('app_apicheck_detail_apirule', {'id': apiCheckID}) }}*. Předávání *id* napříč kontrolerem a šablonou je důležité pro možnost přepínání mezi tabulkami *rule* a *lastApiResponse*. V kontroleru poté dané *id* získám pomocí *$id = $request->attributes->get('id')* a použiji ho při hledání dat v databázi vztahující se k *id* daného *Apichecku* a následné přípravy tabulky. V poslední řadě ho opět předám do šablony.



Obrázek 3: Ukázka stránky podrobněji daného ApiChecku řešené pomocí tabů

Poslední úpravy na tomto projektu pak spočívaly v úpravě front-end. Jednalo se především o nastavení responzivity DataTable. Problém z části řeší balíček sám, je však zapotřebí nastavit limity. Další změny pak spočívaly v úpravách týkajících se vzhledu a vytvoření přihlašovací stránky.

5 Teoretické a praktické znalosti a dovednosti získané v průběhu studia uplatněné studentem v průběhu odborné praxe

Během absolvování odborné praxe jsem se setkal s mnoha technologiemi. Při práci s nimi jsem využil celou řadu získaných znalostí, které jsem získal během studia na VŠB.

Jednou z hlavních technologií, která byla mou náplní bakalářské praxe, je jazyk PHP. Ačkoliv jsem se s jazykem PHP setkal poprvé až zde, v jeho pochopení a používání mi velice pomohla všeobecná znalost jazyků, kterou jsem nabyl vykonáním předmětů v průběhu studia. Velkým přínosem mi byly předměty Programování I a II, které mi představily programovací paradigmata, a především mi poskytly znalosti v oblasti objektově orientovaného programování. Získané teoretické znalosti dále rozšířily navazující předměty: Programovací jazyky I, II a Algoritmy I, II, v nichž jsem si více prohloubil především praktické dovednosti na projektech.

Dalším předmětem, který nemohu opomenout, je Tvorba aplikací pro mobilní zařízení I. Díky tomuto předmětu jsem získal a především rozšířil mé dosavadní znalosti v oblasti HTML, CSS a JavaScriptu. Tyto nabyté teoretické znalosti a praktické dovednosti mi velice ulehčily front-end vývoj během vykonávání praxe. Dalším předmětem, který jsem absolvoval současně s odbornou praxí, byl předmět Vývoj internetových aplikací, který mi prohloubil teoretické znalosti a rozšířil přehled technologiích používaných pro tvorbu internetových aplikací.

6 Znalosti či dovednosti scházející studentovi v průběhu odborné praxe.

Velké nedostatky jsem poznal hned na začátku odborné praxe, a to z důvodu neznalosti GIT verzovacího systému. Díky absolvování odborné praxe jsem zjistil, jakou nedílnou a nepostradatelnou součástí tento nástroj je. Zde si velice vážím pomoci zaměstnanců, kteří mě do tohoto nástroje zasvětili a zbytek jsem doplnil samostudiem. Dle mého názoru by bylo dobré na škole seznámit studenty blíže alespoň s nějakým verzovacím systémem, jelikož je součástí většiny IT firem. Oceňuji předmět Tvorba aplikací pro mobilní zařízení II, kde se do jisté míry pracuje s verzovacím systémem, avšak vykonával jsem ho současně s odbornou praxí, takže jsem znalosti o této technologii už měl.

Dalším nedostatek, se kterým jsem se setkal, byla neznalost jazyka PHP a jeho frameworků Symfony a Nette. Ovšem je mi jasné, že se nedají do náplně studia zahrnout veškeré jazyky a frameworky, které existují. Také je nutné zmínit, že všeobecná příprava, kterou mi poskytla vysoká škola, mi velice urychlila pochopení a používání toho jazyka, jak jsem již zmínil v kapitole 5.

7 Závěr

Absolvování odborné praxe ve firmě Webvalley s.r.o, jako možnost vykonání bakalářské práce, hodnotím velice pozitivně. Tato možnost mě zaujala už při rozhodování a výběru vysoké školy. Jsem rád za tento způsob vypracování závěrečné práce a získané zkušenosti již během studia, které jsou pro mě velkým přínosem. Získané praktické dovednosti a rozšířené teoretické znalosti mi umožnily kariérní růst a jistě najdou v budoucnu uplatnění. Velkým přínosem mi také bylo nahlédnutí do fungování větší firmy, a především seznámení se s prací v týmu a způsob řešení problémů. Dále jsem se naučil pracovat s řadou pro mě nových technologií, jakými jsou například composer nebo verzovací systém Git, jejichž znalost bude velkou výhodou při nástupu do budoucí práce. V neposlední řade jsem získal především praktické zkušenosti s jazykem PHP a frameworkem Symfony, ve kterých jsem byl při nástupu nováčkem. Díky různorodosti úkolů jsem měl možnost vyzkoušet si různé rozšíření jazyku a postupy při řešení různých úkolů, jako jsou například tvorba služeb či administrací.

V průběhu odborné praxe jsem měl možnost pracovat na více projektech, ať už samostatně nebo v týmu, na kterých jsem splnil řadu úkolů. Měl jsem možnost pracovat jak na projektech ve finální verzi, tak na projektech od jejich samotného začátku. Velice oceňuji komunikativnost kolektivu ve firmě Webvalley a především nápomocnost při řešení problémů. Zároveň se mi líbila různorodost úkolů, což mi ukázalo směr, kterým bych se chtěl ubírat v budoucnu, a to konkrétně front-end vývojář.

Literatura

[1] O nás. *Webvalley* [online]. [cit. 2019-01-28]. Dostupné z: https://www.webvalley.cz/o-nas/

[2] *W3Schools Online Web Tutorials: HTML Introduction* [online]. [cit. 2019-02-10]. Dostupné z: https://www.w3schools.com/html/html_intro.asp

[3] *W3Schools Online Web Tutorials: CSS Tutorial* [online]. [cit. 2019-02-11]. Dostupné z: https://www.w3schools.com/css/default.asp

[4] *Composer* [online]. [cit. 2019-02-10]. Dostupné z: https://getcomposer.org/

[5] *Git* [online]. [cit. 2019-02-11]. Dostupné z: https://git-scm.com/

[6] *PHP: Hypertext Preprocessor* [online]. [cit. 2019-02-16]. Dostupné z: https://www.php.net/

[7] *Rychlý a pohodlný vývoj webových aplikací v PHP | Nette Framework: Documentation* [online]. [cit. 2019-02-15]. Dostupné z: https://doc.nette.org/cs/3.0/

[8] *Symfony, High Performance PHP Framework for Web Development* [online]. [cit. 2019-02-22]. Dostupné z: https://symfony.com/

[9] *W3Schools Online Web Tutorials: JavaScript Tutorial* [online]. [cit. 2019-02-15]. Dostupné z: https://www.w3schools.com/js/

[10] *JQuery* [online]. [cit. 2019-02-13]. Dostupné z: https://jquery.com/

[11] Translations (Symfony Docs). *Symfony, High Performance PHP Framework for Web Development* [online]. [cit. 2019-02-18]. Dostupné z: https://symfony.com/doc/current/translation.html

[12] HUJER, Martin. Jaké novinky přinese PHP 7. In: *Zdroják - o tvorbě webových stránek a aplikací* [online]. [cit. 2019-02-19]. Dostupné z: https://www.zdrojak.cz/clanky/jake-novinky-prinese-php-7/

[13] Databases and the Doctrine ORM (Symfony Docs). *Symfony, High Performance PHP Framework for Web Development* [online]. [cit. 2019-02-21]. Dostupné z: https://symfony.com/doc/current/doctrine.html

[14] Forms (Symfony Docs). *Symfony, High Performance PHP Framework for Web Development* [online]. [cit. 2019-02-25]. Dostupné z: https://symfony.com/doc/current/forms.html

[15] Symfony Twig Extensions (Symfony Docs). *Symfony, High Performance PHP Framework for Web Development* [online]. [cit. 2019-03-02]. Dostupné z: https://symfony.com/doc/current/reference/twig_reference.html

[16] The HttpFoundation Component (Symfony Docs). *Symfony, High Performance PHP Framework for Web Development* [online]. [cit. 2019-03-02]. Dostupné z: https://symfony.com/doc/current/components/http_foundation.html

[17] Stwe/DatatablesBundle: This Bundle integrates the jQuery DataTables plugin into your Symfony application. *GitHub* [online]. [cit. 2019-03-08]. Dostupné z: https://github.com/stwe/DatatablesBundle

[18] EntityType Field (Symfony Docs). *Symfony, High Performance PHP Framework for Web Development* [online]. [cit. 2019-03-10]. Dostupné z: https://symfony.com/doc/current/reference/forms/types/entity.html

[19] PHP: cURL - Manual. *PHP: Hypertext Preprocessor* [online]. [cit. 2019-03-27]. Dostupné z: https://www.php.net/manual/en/book.curl.php